PL-TR-96-1155

PL-TR-96-1155

ADA 95 PERSISTENT OBJECTS

Michael T. Rowley, PhD

Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

October 1996

Final Report

PHILLIPS LABORATORY
Space Technology Directorate
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

PL-TR-96-1155

This report has been approved for publication.

[illegible]

STEVEN LINDSAY, Capt, USAF
Project Manager

FOR THE COMMANDER

[illegible]

NANCY L. CROWLEY, Lt Col, USAF
Acting Chief, Space Operations and Simulation Division

[illegible]

CHRISTINE M. ANDERSON
Director, Space Technology Directorate

DRAFT SF 298

1. Report Date (dd-mm-yy) October 1996	2. Report Type Final	3. Dates covered (from... to) 04/96 to 08/96

4. Title & subtitle Ada 95 Persistent Objects	5a. Contract or Grant # F29601-95-C-0003
	5b. Program Element # 62301E
6. Author(s) Michael T. Rowley, PhD	5c. Project # ARPA
	5d. Task # TC
	5e. Work Unit # BB
7. Performing Organization Name & Address Intermetrics, Inc. 733 Concord Avenue Cambridge, MA 02138	8. Performing Organization Report # IRMA 1462
9. Sponsoring/Monitoring Agency Name & Address Phillips Laboratory 3550 Aberdeen Avenue SE Kirtland, AFB, NM 87117-5776	10. Monitor Acronym
	11. Monitor Report # PL-TR-96-1155

12. Distribution/Availability Statement
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

13. Supplementary Notes

14. Abstract
We have defined and implemented a programming environment for developing persistent Ada 95 applications. The mechanism for persistence is new binding to the emerging Object Data Management Group (ODMG) object-oriented database standard. In this report, we describe the binding and the rationale behind it. We also describe our implementation of the binding using the Texas Object Storage Manager as the back-end. We use the binding to develop a large persistent application, and we describe the issues that arose in that development and the performance of the resulting application.

15. Subject Terms Databases, Ada Programming, Object-Oriented Development

Security Classification of			19. Limitation of Abstract	20. # of Pages	21. Responsible Person (Name and Telephone #)
16. Report Unclassified	17. Abstract Unclassified	18. This Page Unclassified	Unlimited	62	Capt Steve Lindsay (505) 846-8986 ext 325

Preface

This report describes the work done by Intermetrics to develop a persistent programming environment for Ada 95. While the work entailed several major tasks, this report concentrates on the motivation and design of our Ada 95 binding to the emerging Object Data Management Group (ODMG) object-oriented database standard. Among the tasks of this effort, the binding design has the largest impact on the programming environment for a persistent application programmer. Much of the effort on the project, however, was spent on the implementation of the binding, using the Texas object storage manager as the back-end. We do allot some of this report to discussing the issues that arose in that implementation. We also describe a large example application that we built with this persistent programming environment, and how the environment simplified its development. Lastly, we describe some benchmarks that we ran, which showed that the persistent Ada application ran only about 10% slower than a non-persistent C implementation of the same program.

CONTENTS

CONTENTS (Continued)

Introduction

Intermetrics' Persistent Object Bindings project undertook to design and implement a technology for creating persistent object-oriented programs using Ada 95. We considered several fundamentally different approaches to adding persistence to an Ada 95 application. Relational databases are fairly well standardized and certainly widely used for persistence. However, we wanted to make it possible to develop persistent applications using a style that is more similar to devloping transient applications than is possible when the data must ultimately be stored in a relational database. Also, there already is a good binding from Ada to relational databases: SAMeDL, an Intermetrics product.

Alternatively, we could have followed the lead of other new languages, such as Eiffel, Java and Sather, which have all provided persistence by releasing a library that can write arbitrary objects to disk. Usually, these libraries are capable of following any pointers from an object so they can store the entire connected graph that starts at some given object. The graph is stored in an address independent representation, so that it may be read in by a later execution of the application. While this provides persistence, it would not be sufficient for any application that needed any kind of database functionality. The data has to be explicitly stored and loaded. There is no concurrency control. The amount of data is limited, and there is no provision for logging and recovery.

Since Ada 95 is an object-oriented language, it seems natural that the best way to create persistent applications with it would be similar to the best techniques for creating persistent applications in other object-oriented languages, namely with object-oriented database systems. Object-oriented databases solve the semantic mismatch problem by closely integrating the database with the programming language, essentially using the programming language as both the data definition language and the data manipulation language for the database. However, as yet, no object-oriented database company has distributed a product based around Ada 95.

To make our work as reusable as possible, we prefer not to just define a binding to a specific object-oriented database system. Instead, we have defined a binding to the ODMG standard. This is a still emerging standard for use by object-oriented database companies. Few companies have released systems that conform to the standard, but almost all of the major object-oriented database vendors have committed to supporting it. The ODMG standard currently defines a standard binding for two languages: C++ and Smalltalk. The two bindings are different in fairly basic ways, so that they may closely resemble the object models of their language environment.

We have defined an Ada95 binding to ODMG, which we believe fits very well with the programming model for Ada. We have tried to define the binding in such a way that it would be possible for any of the companies that develop object-oriented databases to make use of the binding with their product. However, since we are not able to actually implement the binding for each product, we do yet not know for certain whether we achieved that goal.

In addition to defining the binding to ODMG, we also created an implementation of the binding for an object-oriented database called Texas, which is a freely distributed system from the University of Texas at Austin. Texas is not a full featured object-oriented database system, but it does implement persistence in the same style as ObjectStore, the most popular object-oriented database system, and it does implement transactions, large databases, logging and recovery.

Once we had designed and started implementing the ODMG binding using Texas for a backend it became clear that we would need to also develop a substantial persistent application, since benchmarks and toy applications could never be used to determine whether the binding was effective for developing large persistent applications. We settled on developing a MOO server. (MOO is a double acronym, standing for MUD Object-Oriented, where MUD stands for Multi-User Domain).. MOO is an interpreted object-oriented language, similar to Self by David Ungar, but where all of the objects are implicitly persistent. The language has a number of features that are specifically designed for developing the virtual worlds of MUDs.

A MOO system attracted us as a demonstration vehicle for a number of reasons:

- The data needs to be persistent, and recoverable in the case of system crashes;
- It is a fairly complex system, whose implementation benefits greatly from following good software engineering practices;
- It naturally makes use of many of the most important features of Ada, including: a complex type hierarchy of tagged types, overloading, dispatching, tasks, protected types, generic functions, generic packages, etc;
- The system needs to be fast, and there is an existing C implementation of the system to compare its speed against.

By developing the MOO system, we put numerous stresses on both the design and implementation of the ODMG binding. Throughout the project we were able to find ways to improve the usability of the binding.

In the end we developed a complete implementation of MOO, so we were able to compare its performance with LambdaMOO, the existing implementation that was written in C. LambdaMOO has a very crude model of persistence. All of the persistent data is kept in virtual memory at all times. Then, once a day, it goes through all of its data translating it into an ASCII representation that it writes out to disk. The only purpose to writing it to disk is in case of crashes. If it crashes, it rebuilds the entire database based on the last ASCII representation on disk.

However, LambdaMOO is quite fast. It was written in tight C code, that has been continually improved upon over the past several years. Our own version of the server is still in its infancy. There are still several potential optimizations that we are aware of, but that we have not yet implemented. We are also using a version of Gnat (v. 3.05) in which inter-unit function in-lining doesn't work. Since our code is very modularized, we have many small inter-unit calls for simple functions or even just accessor calls. Nonetheless, our current version of MOO, with persistence, is only about 10% slower than LambdaMOO. With only a little more work we think it can be considerably faster.

Since the most important aspect of our work was the development of the ODMG binding design, most of this report will be devoted to describing and rationalizing the choices that were made for that binding. We will then describe the implementation in some detail, and finally we will describe the MOO system that we developed with it.

ODMG Binding For Ada

In this section we describe our specification for an Ada 95 binding to ODMG. After a brief summary of the binding, we will describe in detail: the principles that guided the design of the binding; the persistence mechanism; and, the Ada package specifications for storage managers and collection libraries.

Summary

The Ada binding for the ODMG standard object-oriented database model uses Ada's unique concept of storage pools as the basis for a persistence mechanism that is independent of type. Unlike the standard C++ binding, which requires that all persistence capable types have a common ancestor type, the Ada binding allows any type to be made persistent. This additional flexibility allows existing types to be made persistent without modifying their inheritance hierarchy. This is especially useful for types which are very deep in an inheritance hierarchy, where you don't want to make the whole hierarchy persistent.

The persistence of an object is determined by the storage pool that manages it. Each heap allocated object in Ada is allocated within a storage pool. The storage pool used is determined by the access type (pointer type) that is used for the new object. The storage pool for an access type handles the allocation and deallocation of its instances. By using separate storage pools for each access type, the compiler may use a different representation of pointers depending on the storage pool. With this facility, it is possible to use persistent object identifiers (OIDs) or some other indirect reference as the pointer type for persistent objects without changing the way these objects are accessed in the code.

The current storage pool mechanism is not quite strong enough to handle persistent objects for many of the most common object-oriented database systems, since there is no hook for dereferencing access values. While some object-oriented databases, like POET and ObjectStore, use pointer *swizzling* where the in-memory representation of an object identifier is just a pointer, most of the other object-oriented databases have a representation of object identifiers that is not just a direct pointer to them.

In order to support non-swizzling object-oriented database systems, we define a new language attribute for specifying the *storage manager* of an access type. The storage manager type is an abstract type that is an extension of the storage pool type. In addition to the storage pool operations, it also defines operations for dereferencing an access value whose type uses that storage manager. While the storage manager attribute was designed for handling persistent objects with object-oriented databases, it is general enough to be used for other memory management purposes, such as garbage collected storage pools, which are difficult to implement with just the facilities available in the storage pool type.

Other aspects of the binding are more similar to other object-oriented language bindings. A one-to-many relationship is represented by an attribute for the to-one direction, and a collection object for the to-many direction. Transactions and databases are represented as abstract data types that correspond very closely to their C++ class counterparts. The collection and iterator types are also very similar.

The representation of a schema in this binding is straight forward. The Ada specification of each package declares which of its access types will be persistent by specifying that those types use ODB_Persistent_Manager as their storage manager. The persistent types defined in individual package specifications are combined into *application schemas*, which contain all of the persistent types defined by all of the packages linked into that application. *Database schemas* define the types that are, or might be stored within an individual database. They are not separately specified, but are defined by the union of all application schemas of the applications that write to the database. This is actually a larger schema than strictly necessary, since not all of the persistent types in an application are stored in all databases used by that application, but it is a simple conservative mechanism.

Since this binding requires so little special treatment for adding persistence, it is easy to modify an existing transient application to become persistent. The pointer types for the persistent data just need to be declared to use the persistent storage manager, and some code has to be added so that databases are opened and transactions are started and committed. Because this transition is so simple, it is not only possible to convert entire applications to persistent applications, but it also makes it easier to develop new applications, because of all of the code that can be reused from libraries designed for use in transient applications.

Binding Design Principles

The binding was designed with the following principles:

Fits well with the Ada programming style

A program that uses persistent data should not be very much different from a well designed Ada program that uses only transient data. Persistence constructs that work well with other languages may be possible in Ada, but if they are not natural, better alternatives should be found.

Easy to make existing code persistent

It should be easy to make code that was written to be used with transient data, into one that uses persistent data. While this is useful for transforming entire applications, the more important case is for existing libraries. Ada is a language that is designed to provide much of its power to the programmer by allowing them to reuse existing code libraries, rather than code from scratch. So even a new persistent application will need to make persistent objects whose types are defined in preexisting libraries.

Easy to implement for any of the ODMG member companies

No matter how elegant the Ada persistence model is, it is worthless as a standard if few vendors are willing or able to use it with their ODB products. The implementation does not have to be trivial for all database systems, but it should be feasible.

There are two major models for existing ODB systems: using hooks to the virtual memory system to automatically catch references to uncached objects and then load the objects and swizzle all their references to pointers; or, by using smart references explicitly in programs, with help from the language to make the use of references look similar to the use of pointers. Each of these object database architectures should be able to be used as the implementation for the Ada binding.

Follows the ODMG Object Model fairly closely

Looking at differences between the C++ and Smalltalk bindings to ODMG, it is clear that there is a lot of latitude available to the language binding designer. If such a designer decided that they did not agree with the decisions made by ODMG, it would be possible to create a persistence model that looked nothing like the ODMG object model and then define a mapping from the Ada model to the ODMG model. We prefer to accept the fundamental concepts of the model as they are.

The most important issue related to this tradeoff is the way relationships are handled. Rather than use a separate "relation" construct, as is used by object-relational systems, relationships are represented by pointers and collections, depending on whether there is one or many objects respectively. This is a fundamental characteristic of the ODMG model and so choosing a different representation would not be appropriate in an ODMG binding.

Requires no persistent ancestor type

There are great advantages to not requiring a common ancestor for persistence. The most important advantage is that it is not necessary to modify the inheritance hierarchy to make objects from a preexisting library persistent. In a language without multiple inheritance, like Ada, making capabilities available only through inheritance is dangerous. As soon as you have to use two such capabilities together that each require their own common ancestor, you get in trouble.

Another advantage is that by not requiring a persistent ancestor type it is possible to make objects of non-tagged types persistent. Many types in a well designed Ada program will not need to be tagged. It should not be necessary to make them tagged just so that they may be persistent.

Without a common ancestor type, it is not possible to define operations that are automatically callable for all persistent capable objects. There are several such functions in the ODMG model: name functions, the lookup function, and query functions. These are handled in the Ada binding with as generic functions.

Ada Binding

Storage Pools

Ada is fairly unusual in the fact that two access types that both access the same designated type are not freely interchangeable. You cannot legally assign an access value to any access variable with the same designated type. For example:

```
type X_Ptr is access X;
type X_Ptr2 is access X;
procedure P is
   a: X_Ptr;
   b: X_Ptr2;
begin
   a := new X;
   b := a;  -- Illegal code.  'a' and 'b' are of different access types.
end;
```

In the above code, X_Ptr and X_Ptr2 each has their own storage pool. Any new object created for an access variable of type X_Ptr will be allocated using the Allocate routine of the storage pool object assigned to X_Ptr. Similarly, X_Ptr2's storage pool is used to create new objects for X_Ptr2 access variables (see Figure 1).

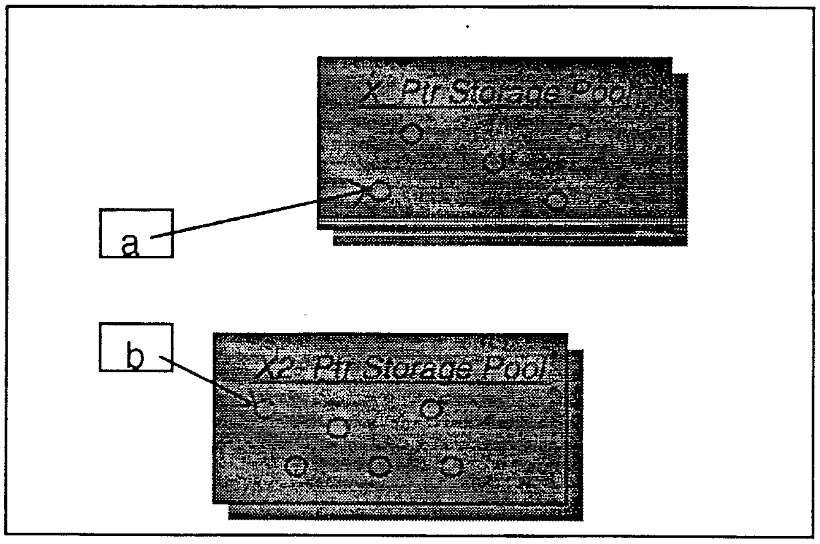

Figure 1

The storage pool for an access type is specified by the 'Storage_Pool attribute. A storage pool can be any object whose type is a descendent of Root_Storage_Pool. Root_Storage_Pool is an abstract type that defines an interface that includes functions for allocation, deallocation and determining and setting the storage size of the pool. The specification is as follows:

```
with Ada.Finalization;
with System.Storage_Elements;

package System.Storage_Pools is

   type Root_Storage_Pool is abstract
     new Ada.Finalization.Limited_Controlled with private;

   procedure Allocate
     (Pool                        : in out Root_Storage_Pool;
      Storage_Address            : out System.Address;
      Size_In_Storage_Elements : in System.Storage_Elements.Storage_Count;
      Alignment                  : in System.Storage_Elements.Storage_Count)
   is abstract;

   procedure Deallocate
     (Pool                       : in out Root_Storage_Pool;
      Storage_Address            : in System.Address;
      Size_In_Storage_Elements : in System.Storage_Elements.Storage_Count;
      Alignment                  : in System.Storage_Elements.Storage_Count)
   is abstract;

   function Storage_Size
     (Pool : Root_Storage_Pool)
      return System.Storage_Elements.Storage_Count
   is abstract;

private

   type Root_Storage_Pool is abstract
     new Ada.Finalization.Limited_Controlled with null record;

end Storage_Pools;
```

To create a specialized storage pool it is only necessary to create a new type that inherits from Root_Storage_Pool, then define the functions for it. Assume that you create such a type, called My_Storage_Pool. You then must declare an object of that type, and then assign it to the appropriate access types using the 'Storage_Pool attribute. For example:

```
My_Pool: My_Storage_Pool;

type X_Ptr is access X;
for X_Ptr'Storage_Pool use My_Pool;
```

Then any object that is allocated on the heap for assignment into a variable or parameter of type X_Ptr will be allocated using the routines from My_Pool.

Storage Managers

Since access values are not interchangeable, it should be possible to use different representations for different access types. Unfortunately, the storage pool mechanism isn't strong enough to do that. The compiler assumes that access types with user defined storage pools produce values that are the addresses of the objects being referenced. In order to handle persistent access values, it must be possible to specify a user-defined procedure for dereferencing.

Another difference between storage managers and storage pools is that storage manager access values must be able to be larger than regular access values. If a persistent access value is one of the fields of some object, that field needs to be large enough to hold a persistent OID. There is no language imposed reason why all access values in Ada have to be the same size, and in fact, some compilers use different sizes for different kinds of access values.

For storage manager access values, the space used for their representation is specified by a generic parameter to the storage manager package. So the beginning of the storage manager package specification is:

```
generic
   type Access_Type is private;
package Storage_Manager is
   type Root_Storage_Manager is abstract
     new Ada.Finalization.Limited_Controlled with private;

   -- Specifications for Dereference and the other storage manager procs
   -- ...
end Storage manager;
```

The routines for storage managers will use Access_Type when referring to managed access values and System.Address when referring to addresses. This is in contrast with the storage pool package, which uses System.Address as the type for access values. Each of the storage manager procedures will be discussed separately in the sections below, followed at the end by the complete package specification.

Potential Alternative: Storage manager addresses could be the same size as other addresses, and all of the routines could take and return System.Address. For a persistent storage manager, the address would be a pointer to a cached object descriptor, which would include the OID, plus the address of the object in memory. The problem with this approach is that it does not allocate enough space for the access values for when they are written to disk. At both activation and deactivation time, the object will need to be converted between the disk representation to the in-memory representation.

Dereference

The dereference procedure is called by the compiler to turn an access value into the machine address of the object. For most access types, the access value *is* the address, and so it would be trivial. However, with this hook it is possible for the user to create indirect addressing schemes that are used the same as regular access values. The signature for the dereference procedure is:

```
procedure Dereference(Manager         : in out Root_Storage_Manager;
                      Managed_Address : in     Access_Type;
                      General_Address : out    System.Address;
                      For_Modification: Boolean) is abstract;
```

Like storage pools, storage managers are abstract tagged records. All of the storage manager's routines take the storage manager as the first argument. The Managed_Address argument is the access value to be converted. Its type is "Access_Type", which is a generic formal parameter to the Storage Manager class. General_Address is the output of the procedure that is the machine address of the object in the form that the compiler expects, which is usually the address of the first field of the record.

Dereference also takes a Boolean argument called For_Modification. Many ODBs need to know whenever an object changes, so that they may write the changed objects to disk at commit time. The C++ binding requires a call to a routine called Mark_Modified. This binding attempts to make the introduction of persistence have as little impact as possible. One way to eliminate the need for users to call Mark_Modified is to take advantage of the fact that the compiler knows when the object is being dereferenced is the target of an assignment, or when one of its fields is the target of an assignment. If it is, then the For_Modification argument is passed as True, and if the database requires it, the object is marked as having been modified. An argument that is passed as "in out" can be assumed to be the target of an assignment.

I expect that it should always be possible for only one object to be "For_Modification" in any given assignment. Even if an assignment involves multiple dots, as in "a.b.c.d := x", only one of those objects (the one represented by a.b.c) will be modified.

Allocate

The Allocate routine is very similar to the storage pool Allocate procedure. Unfortunately it cannot be the same. The storage pool Allocate function is given only the amount of space to allocate and the alignment required. The Allocate routine for a persistent storage manager will be implemented as a call to an ODB's object creation routine, and so it is necessary to know the type of object being created. There is no standard encoding for representing any type at run-time. Tagged types can be represented by their tag values, but we explicitly want to allow non-tagged types to be persistent. So the only way to specify the type is by its completely qualified name. Completely qualified means that all package and child package names are included in the name. Here is the specification for the storage manager Allocate function:

```
procedure Allocate
   (Manager                  : in out Root_Storage_Manager;
    Storage_Address          : out Access_Type;
    Size_In_Storage_Elements : in System.Storage_Elements.Storage_Count;
    Alignment                : in System.Storage_Elements.Storage_Count;
    Type_Name                : in String) is abstract;
```

The storage size and the alignment are still given to allow for the possibility that an Allocate routine could be written that was independent of the compiler, and could use those arguments to provide the right amount of space.

The result of Allocate is a value of the generic parameter Access_Type.

Potential Alternative: It would be possible to avoid providing a type name to Allocate if for every managed access type a new storage manager object was created. Since each storage manager would be for only one type, there would be no question of which type is being created by a call to Allocate. The problems with this is that it requires so many storage managers to be created, and it may make it harder to put objects of different types together on the same page.

Rereference

In addition to dereferencing an object, it is also sometimes necessary to "rereference" an object, which means convert from an address to an access value. Procedure parameters do not need to declare an explicit access type, but may simply use the form "access X". This is called an anonymous access type. It must be able to take any access value with designated type X. The simplest way for the compiler to implement passing an persistent access value as such a parameter would be to dereference it first, and then pass the address as the parameter.

It must be possible to convert from an anonymous access value back to its original access type. For a persistent access type this will mean converting from an address back to the persistent access value. For such conversions there is the Rereference procedure, which has the following signature:

```
procedure Rereference(Manager         : in out Root_Storage_Manager;
                      General_Address : in     System.Address;
                      Managed_Address : out    Access_Type;
                      Type_Name       : in     String) is abstract;
```

If General_Address is not the address of an object allocated by this storage manager, then it should return a null reference. This means that it must be possible for a storage manager to determine whether an address is one that was allocated by it, and if it was, get back the managed access value for it.

Potential Alternative: Anonymous access parameters could be handled without converting to an address. Instead pass pointer to the persistent access value, plus the storage manager that knows how to dereference it. The compiler would use the passed in storage manager whenever it needs to dereference.

Potential Addition: In addition to Rereference, it may be desirable to have a Convert function, which takes an address of an object which was *not* allocated by this storage manager and converts it to a managed address. For a persistent storage manager this would mean making the transient object persistent. One problem with this is that it would be very hard to implement Rereference if the storage manager could not depend on the fact that all rereferencable objects were originally created by that storage manager, where that storage manager was able to add additional overhead data adjacent to the allocated space for the object.

Complete Storage Manager Definition

```
generic
   type Access_Type is private;
package System.Storage_Manager is

   type Root_Storage_Manager is abstract
     new Ada.Finalization.Limited_Controlled with private;

   procedure Allocate
     (Manager                    : in out Root_Storage_Manager;
      Storage_Address            : out Access_Type;
      Size_In_Storage_Elements   : in System.Storage_Elements.Storage_Count;
      Alignment                  : in System.Storage_Elements.Storage_Count;
      Type_Name                  : in String)
   is abstract;

   procedure Deallocate
     (Manager                       : in out Root_Storage_Manager;
      Storage_Address            : in System.Address;
      Size_In_Storage_Elements   : in System.Storage_Elements.Storage_Count;
      Alignment                  : in System.Storage_Elements.Storage_Count)
   is abstract;

   procedure Dereference(Manager          : in out Root_Storage_Manager;
                         Managed_Address  : in Access_Type;
                         General_Address  : out    System.Address;
                         For_Modification: Boolean) is abstract;

   procedure Rereference(Manager          : in out Root_Storage_Manager;
                         General_Address  : in     System.Address;
                         Managed_Address  : out    Access_Type;
                         Type_Name        : in     String) is abstract;

private
   type Root_Storage_Manager is abstract
     new Ada.Finalization.Limited_Controlled with null record;

end System.Storage_Manager;
```

Package ODB

The storage manager package and attribute is a general mechanism that could be used for other things in addition to adding persistence. For example, it could be used to create garbage collected storage managers (a brief description of this possibility is discussed later). As such we have defined it as a child package of system.

All of the types and functions specific to the ODMG binding are in a single package called ODB. It contains the specifications for: the persistent storage manager, collection types, the transaction and database types, and some general generic operations. The collection types are defined as child packages. All of these are combined into one package, since although it is fairly large, an application will always need almost all of it. There is no point in dividing it into smaller packages, when most applications would have to "with" all of the smaller packages.

We will discuss each of the major abstract data types that are included in the ODB package in the following sections.

Persistent Storage Manager

To create a persistent storage manager, it is necessary to define the type that will be used for persistent access types, then create a new subtype of Root_Storage_Manager that implements all of the storage manager operations. There are then some additional operations that are only appropriate for the persistent storage managers. The definition of the Persistent_Storage_Manager, as found in the ODB package follows. The exact type definition for Persistent_Access_Type is elided, as is the signatures for the routines that are inherited from Root_Storage_Manager. Unfortunately, the definition of Persistent_Access_Type cannot be private, since it must be used as a generic parameter to the instantiation of the public Storage_Manager_Pkg.

```
type Persistent_Access_Type is ...
package Storage_Manager_Pkg is new Storage_Manager(Persistent_Access_Type);

type Persistent_Storage_Manager is
  new Storage_Manager_Pkg.Root_Storage_Manager with private;

procedure Allocate(Manager: in out Persistent_Storage_Manager;  -- ...
procedure Deallocate(Manager: in out Persistent_Storage_Manager; -- ...
procedure Dereference(Manager: in out Persistent_Storage_Manager;  -- ...
procedure Rereference(Manager: in out Persistent_Storage_Manager; -- ...

function Default_Db(Manager     : Persistent_Storage_Manager)
                    return ODB.Database;
procedure Set_Default_Db(Manager: in out Persistent_Storage_Manager;
                         Db     :  ODB.Database);
```

It is necessary to instantiate the Storage_Manager_Pkg before defining the Persistent_Storage_Manager type as a child of Root_Storage_Manager, since the persistent storage manager will need to use the Persistent_Access_Type in its implementation of the inherited routines.

There is one additional attribute on a persistent storage manager that doesn't exist for other storage managers, that is the Default_Db. As with all changeable attributes, it has an accessor function and a set function. The purpose of the attribute is to define the database that newly allocated objects should be created in. The ODB that is used to implement the persistent storage manager needs to be passed the desired database for new objects.

The C++ binding determines the database to create new objects in by adding a new database parameter to "new". Ada does not have a mechanism for passing arguments to new. Instead, the programmer sets the Default_Db for the storage manager, and that is used for all new objects until it is changed with a Set_Default_Db. Opening a database automatically causes it to become the default database, so this routine is only used by applications that create objects in more than one database at a time. Whether this is more convenient or less convenient than specifying the database within each new command depends on the application. Many application create several objects at a time in the same database. For these applications, having a default database is more convenient. Only for applications that use multiple databases truly simultaneously could this technique be inconvenient, due to the fact that many of the calls to new would have to be preceded by calls to Set_Default_Db.

One possible area of confusion with the storage manager mechanism is the fact that a storage manager may be thought by some people as corresponding to a database, but this is not accurate. Objects from multiple databases are handled by the same storage manager. This is necessary, since storage managers are specified at compile-time. It must be possible to open and and use several different databases with the same code, and therefore with the same variables and storage managers.

Database

The Database type is fairly similar to the C++ Database class. Database objects of this type are transient. Databases cannot be created programatically. By opening a database it automatically becomes the default database for future creations of persistent objects.

Like the C++ binding, the Ada binding groups all of the functions related to object naming with the database type. The difference is that in Ada, there is no one type that can be used to represent any persistent capable type, since we do not have such a common ancestor. Instead, there are generic functions for these routines. Before using any of these routines, the user must first instantiate them. Most types will not need the name functions, so these instantiations are not generated for every type.

There is one additional function that is here rather than by itself, as it is in the C++ binding, and that is Oql. Since the query will operate on a single database, we have put it with the other database operations. The specification for the Database type follows:

```
type Access_Status is (Not_Open, Read_Write, Read_Only, Exclusive);
type Database is limited private;

procedure Open(Self  : in out Database;
               Name  : String;
               Status: Access_Status := Read_Write);
procedure Close(Self: in out Database);

generic
   type Object_Type is limited private;
   type Object_Ptr is access Object_Type;
procedure Set_Object_Name(Self   : in out Database;
                          Object: Object_Ptr;
                          Name   : String);
generic
```

```
   type Object_Type is limited private;
   type Object_Ptr is access Object_Type;
function Get_Object_Name(Object: Object_Ptr) return String;

generic
   type Object_Type is limited private;
   type Object_Ptr is access Object_Type;
function Lookup_Object(Self  : Database;
                       Name  : String)
                       return Object_Ptr;

procedure Rename_Object(Self    : in out Database;
                        Old_Name: String;
                        New_Name: String);

generic
   type Object_Type is tagged limited private;
   type Object_Ptr is access all Object_Type'class;
function  Oql(Query: String) return Object_Ptr;
```

Transaction

The transaction type is the most similar to its C++ transaction class counterpart. The transaction semantics are the same, and as in C++, all operations on persistent data must happen within a valid transaction. Because transactions are not intended to be inherited from the transaction type is not tagged. Its definition follows:

```
type Transaction is limited private;

procedure Begin_Transaction     (Self: in out Transaction);
procedure Commit_Transaction    (Self: in out Transaction);
procedure Abort_Transaction     (Self: in out Transaction);
procedure Checkpoint_Transaction(Self: in out Transaction);
```

Collection and Iterator Packages

The collection packages are child packages of ODL. There is a generic Collection package, which defines a Collection type, followed by generic packages for List, Bag, Set, and Varray, which all define corresponding types. In order to create any child of Collection for some type (e.g. List of Person), it is necessary to first instantiate the

Collection package for Person, and then instantiate List passing the Collection package as a generic parameter; e.g.:

```
package Person_Collection is new Odb.Collection_Package(Person_Ptr);
package Person_List is new ODB.List_Package(Person_Collection);
```

If a different type of collection is then desired for the same object type, the instantiation should use the same collection type, e.g.:

```
package Person_Set is new ODB.Set_Package(Person_Collection);
```

In this way it is possible for procedures that don't really care what implementation is used for the person collection they want to operate on to declare their parameter as Person_Collection, and be able to take instances of Person_Set or Person_Collection.

In order to make this kind of substitutability convenient, we had to introduce a new parent type for Collection, which we call Collection_Ancestor. It contains all of the Collection routines that do not take a parameter of the generic element type, and so it is not defined in a generic package. If there were no Collection_Ancestor type and all of its routines were put in the Collection package, the it would not be possible "use" more than one collection package, even if they were defined on different element types. The routines which use the element type as a parameter or return value would not conflict because of the Ada overloading rules, but any routine that does not use the element type would be seen to the compiler as having two definitions. Therefore all of those routines are defined in the Collection_Ancestor type. This way, dispatching rather than overloading can differentiate between the routines. This is the Collection_Ancestor type:

```
type Collection_Ancestor is abstract new Ada.Finalization.Controlled
   with private;

function Cardinality(Self: Collection_Ancestor) return Integer is abstract;
function Is_Empty   (Self: Collection_Ancestor) return Boolean is abstract;
function Is_Ordered (Self: Collection_Ancestor) return Boolean is abstract;
function Allows_Duplicates (Self: Collection_Ancestor) return Boolean
   is abstract;
function Exists_Element(Self: Collection_Ancestor;
                        Predicate: String)      return Boolean is abstract;
procedure Remove_All (Self: in out Collection_Ancestor) is abstract;
```

The same problem with Collection, also occurs with Iterator. There is an Iterator_Ancestor type that defines all of the routines that would not use the generic type parameter in the generic Iterator package. It is as follows:

```
type Iterator_Ancestor is abstract tagged null record;
function  Not_Done(Self: Iterator_Ancestor) return Boolean is abstract;
procedure Advance (Self: in out Iterator_Ancestor) is abstract;
procedure Reset   (Self: in out Iterator_Ancestor) is abstract;
```

The generic Collection package takes a single limited private generic parameter. There is no requirement that the collections be instantiated only with persistent access types, or even any kind of access type. You can create a collection of any type. If the collection contains actual objects (not pointers), or if it contains persistent access values for objects, then it may be persistent. It is an error to create a persistent collection of non-persistent access values.

The generic Collection package defines both the Iterator and the Collection abstract tagged types. One noteworthy aspect of the collection types is that, unlike their C++ counterparts, they include routines that convert some or all of the collection to or from Ada arrays. Arrays are a fundamental type in Ada, unlike C++. It is also possible to pass or return an array by value, unlike C++. It is therefore convenient to have routines that allow programmers to deal with arrays. In fact, instead of using iterators to iterate through some or all of a collection, in our experience, it is more common to get the array representation the relevant part of the collection, and then iterate through the array. A loop of the form:

```
      ...
      A: Element_Array := As_Array(List);
   begin
      for I in A'Range loop
          f(A(i));
      end loop;
   end;
```

is more natural than the iterator version, which would be:

```
      Iter: List_Iterator := Create_Iterator(List);
   begin
      while Not_Done(Iter) loop
          f(Get_Element(Iter));
          Advance(Iter);
      end loop;
```

The package specification is as follows:

```
generic
   type T is limited private;
package Collection_Package is

   Index_Out_Of_Bounds          : exception;
   Element_Not_Found            : exception;
   Iterator_Done                : exception;
   Iter_Not_For_This_Collection : exception;

   type Iterator is abstract new Iterator_Ancestor with null record;

   function Get_Element ( Self : Iterator ) return T is abstract;

   type Collection is abstract new Collection_Ancestor with null record;
   type Collection_Ptr is access Collection'CLASS;

     -- functions inherited from Collection_Ancestor

   procedure Copy (
       Self            : access Collection;
       Other           : access Collection
   ) is abstract;
       -- Copy is NOT part of the ODMG spec.
   procedure Delete (
       Self : access Collection
   ) is abstract;
       -- Delete is NOT part of the ODMG spec.
   function "=" (
       Self  : Collection;
       Other : Collection
   ) return Boolean is abstract;
   function Cardinality ( Self : access Collection ) return Integer
       is abstract;
   function Is_Empty ( Self : access Collection ) return Boolean
       is abstract;
   function Is_Ordered ( Self : access Collection ) return Boolean
       is abstract;
   function Allows_Duplicates ( Self : access Collection )
       return Boolean is abstract;
   procedure Remove_All ( Self : access Collection ) is abstract;

     -- Abstract types and routines for all collections

   type Element_Array is array (Integer range <>) of T;
```

```
      function Contains_Element ( Self : access Collection; E : T )
          return Boolean is abstract;
      procedure Insert_Element ( Self : access Collection; E : T )
          is abstract;
      procedure Remove_Element ( Self : access Collection; E : T )
          is abstract;

         -- Routines that use collections and iterators

      function Create_Iterator ( Self : access Collection )
          return Iterator'CLASS is abstract;
      function Retrieve_Element_At (Self : access Collection;
                                    Iter : Iterator'CLASS)
                                    return T is abstract;
      procedure Remove_Element_At (Self : access Collection;
                                   Iter : in out Iterator'CLASS)
          is abstract;
      procedure Replace_Element_At (Self : access Collection;
                                    E    : T;
                                    Iter : in out Iterator'CLASS)
          is abstract;

         -- Routines that use Ada arrays

      function As_Array(Self: access Collection) return Element_Array
          is abstract;
      function Retrieve_Array_At (Self        : access Collection;
                                  Start_Index : Integer;
                                  End_Index   : Integer)
                                 return Element_Array is abstract;
      procedure Insert_Array_At (Self  : access Collection;
                                 Other : access Collection;
                                 Index : Integer)
          is abstract;

          -- Routines that take a predicate

      function  Select_Element(Self: access Collection;
                               Predicate: String) return T is abstract;
          -- Select_Collection replaces Select from the ODMG standard
          -- because select is an Ada keyword.
      function  Select_Collection(Self: access Collection;
                                  Predicate: String)
                                  return Collection is abstract;

      function Oql(S: String) return Collection is abstract;

   end Collection_Package;
```

Subtypes of Collection

```
generic
   with package Collection_Pkg is
      new Collection_Package(<>);
   use Collection_Pkg;
package Set_Package is
```

```
-------- SET ITERATOR ---------

type Set_Iterator is new Iterator with private;

function  Not_Done(Self: Set_Iterator) return Boolean;
procedure Advance (Self: in out Set_Iterator);
procedure Reset   (Self: in out Set_Iterator);
function  Get_Element(self: Set_Iterator) return T;

-- functions inherited from controlled
-- these replace create, delete, and copy from the odmg

procedure Initialize(Self: in out Set_Iterator);
procedure Adjust    (Self: in out Set_Iterator);
procedure Finalize  (Self: in out Set_Iterator);


-------- SET ---------

type Set is new Collection with private;
type Set_Ptr is access Set'CLASS;

-- functions specific to Sets

function  Union       (S1, S2: access Set) return Set_Ptr;
function  Intersection(S1, S2: access Set) return Set_Ptr;
function  Difference  (S1, S2: access Set) return Set_Ptr;

procedure Union_With     (Self: access Set; Other: access Set);
procedure Intersect_With (Self: access Set; Other: access Set);
procedure Difference_With(Self: access Set; Other: access Set);

function Is_Subset_Of       (Self: access Set; Other: access Set)
    return Boolean;
function Is_Superset_Of     (Self: access Set; Other: access Set)
    return Boolean;
function Is_Proper_Subset_Of(Self: access Set; Other: access Set)
    return Boolean;

-- functions inherited from collection_anestor

function "="         (Self: access Set;
                      Other: access Set) return Boolean;
function Cardinality(Self: access Set) return Integer;
function Is_Empty   (Self: access Set) return Boolean;
function Is_Ordered (Self: access Set) return Boolean;
function Allows_Duplicates (Self: access Set) return Boolean;
function Exists_Element(Self: access Set;
                        Predicate: String)    return Boolean;
procedure Remove_All (Self: access Set);

-- functions inherited from Collection

function  Create_Iterator(Self: access Set)
                          return Iterator'Class;
function  Contains_Element(Self: access Set; E: T)
                           return Boolean;
procedure Insert_Element(Self: access Set; E: T);
procedure Remove_Element(Self: access Set; E: T);
function As_Array(Self: access Set) return Collection_Pkg.Element_Array
       is abstract;
```

```
   function Retrieve_Array_At (Self        : access Set;
                               Start_Index : Integer;
                               End_Index   : Integer)
       return Collection_Pkg.Element_Array;
   procedure Insert_Array_At (Self  : access Set;
                              Other : Collection_Pkg.Element_Array;
                              Index : Integer);
   function  Select_Element(Self: access Set; Predicate: String) return T;
   function  Select_Collection(Self: access Set; Predicate: String)
       return Set_Ptr;
   function  Oql(Self: String) return Set_Ptr;

   procedure Initialize(Self: access Set);
   procedure Adjust    (Self: access Set);
   procedure Finalize  (Self: access Set);

private
end Set_Package;
```

```
generic
   with package Collection_Pkg is
      new Collection_Package(<>);
   use Collection_Pkg;
package List_Package is

   -------- LIST ITERATOR ---------

   type List_Iterator is new Iterator with private;

   function  Not_Done(Self: List_Iterator) return Boolean;
   procedure Advance (Self: in out List_Iterator);
   procedure Reset   (Self: in out List_Iterator);
   function  Get_Element(self: List_Iterator) return T;

   -- functions inherited from controlled
   -- these replace create, delete, and copy from the odmg

   procedure Initialize(Self: in out List_Iterator);
   procedure Adjust    (Self: in out List_Iterator);
   procedure Finalize  (Self: in out List_Iterator);

   -------- LIST ---------

   type List is new Collection with private;

   -- functions unique to lists

   function  Retrieve_First_Element(Self: List) return T;
   function  Retrieve_Last_Element(Self: List) return T;
   function  Find_Element         (Self: List; E: T; Start_Pos: Integer)
                                   return Integer;
   function  Retrieve_Element_At  (Self: List; Pos: Integer) return T;
   procedure Remove_Element_At    (Self: in out List; Pos: Integer);
   procedure Remove_Element_At    (Self: in out List; E: T; Pos:  Integer);
   procedure Insert_Element_First (Self: in out List; E: T);
   procedure Insert_Element_Last  (Self: in out List; E: T);
   procedure Insert_Element_After (Self: in out List; E: T; Pos: Integer);
   procedure Insert_Element_Before(Self: in out List; E: T; Pos: Integer);
```

```
    function  Concat(Self: List; Other: List) return List;
    function  "+"    (Self: List; Other: List) return List;
    procedure Append(Self: in out List; Other: List);

    -- functions inherited from collection_anestor

    function "="         (Self: List;
                          Other: List) return Boolean;
    function Cardinality(Self: List) return Integer;
    function Is_Empty    (Self: List) return Boolean;
    function Is_Ordered  (Self: List) return Boolean;
    function Allows_Duplicates (Self: List) return Boolean;
    function Exists_Element(Self: List;
                             Predicate: String)      return Boolean;
    procedure Remove_All (Self: in out List);

    -- functions inherited from Collection

    function  Create_Iterator(Self: List)
                              return Iterator'Class;
    function  Contains_Element(Self: List; E: T)
                               return Boolean;
    procedure Insert_Element(Self: in out List; E: T);
    procedure Remove_Element(Self: in out List; E: T);
    function As_Array(Self: access List) return Collection_Pkg.Element_Array
           is abstract;
    function Retrieve_Array_At (Self        : access List;
                                Start_Index : Integer;
                                End_Index   : Integer)
        return Collection_Pkg.Element_Array;
    procedure Replace_Array_At (Self  : access List;
                                Other : Collection_Pkg.Element_Array;
                                Index : Integer);
    procedure Insert_Array_At (Self  : access List;
                               Other : Collection_Pkg.Element_Array;
                               Index : Integer);

    function  Select_Element(Self: List; Predicate: String) return T;
    function  Select_Collection(Self: List; Predicate: String) return List;
    function  Oql(Self: String) return List;

    procedure Initialize(Self: in out List);
    procedure Adjust    (Self: in out List);
    procedure Finalize  (Self: in out List);

private
end List_Package;
```

Using Storage Manager for Garbage Collection

Even though it does not have a direct bearing on the ODMG binding, we wanted to briefly describe another important use of the new storage manager attribute for access types: garbage collection.

Just as the mechanism available for creating user defined storage pools is not powerful enough to use for persistence, it is also not powerful enough for creating storage pools with garbage collection. A garbage collector needs to be able to either find or keep track of all of the valid access values. This means that the garbage collector needs to know when each of the following occur: a new access variable is initialized with an access value; there is an assignment to an access variable; or when an access variable goes out of scope, and is destroyed. Note that these do not correspond to Allocate and Deallocate from the storage manager, since the garbage collector cares about creation and destruction of access values, not the objects themselves.

Controlled types are Ada mechanism for getting control of Initialize, Adjust (for assignment), and Finalize (for destruction). The problem is that, without storage manager, it is not possible to define these routines for access types, only for tagged record types. What the storage manager provides is the ability to define your own type that will be used to represent access values. If this type is a tagged record that inherits from Controlled, then the appropriate hooks will be called when access values are initialized, assigned or destroyed. The Dereference routine could also be useful to a garbage collector, if it needs to add a level of indirection for the access values.

Generating the Schema Specification

With this binding, it is intended that it should be possible for the schema to be defined in the language neutral ODL. A separate tool can then be used to generate an Ada specification and most of the bodies for the objects in the schema. The Ada schema code generator follows many conventions that we believe result in code that is modular, easy to understand, and easy to modify. As with the C++ binding, the operations are not really important in the ODL, since, unlike with an ORB, the calling operations on objects does not involve the intervention of the ODB. The operation calling mechanism is purely defined by the language.

Mutually referential types

One of the conventions followed by the generator, but not required for persistence, is putting all of the schema object definitions in the same package. The reason for this is that circular reference chains among types are common in object-oriented database schemas. The most common circularities are bidirectional relationships. Each type in the relationship needs to reference the other type. If the two types are in different packages such a relationship can't be defined, due to Ada's restriction against mutually referential package specifications.

However, within a single package, the circularity can be avoided. We define all of the object types as private tagged records at the top of the schema specification file. This allows all of the functions below to be generated without worrying whether all of the parameter types have been defined yet.

Extents

Extents are not explicitly maintained by the database. As in the C++ binding, extents are represented by collections that hold every instance of a type, with newly created objects being added to the extent collection, and deleted objects being removed. Class extents are named objects in the database, so that they may be retrieved as root objects.

One difference from the C++ binding is that there are no constructors in Ada. Instead, any routine can return a new object. The Ada schema code generator uses the convention that all types get a create function. The create function takes parameters to be used to initialize the object fields and adds the new object to its extent if necessary.

One problem with this technique is that it is not guaranteed to add every instance of a type to that type's extent, since there is no way to prevent objects from being created without using the generated create function.

Potential Alternatives: New objects could be added to extents using either the storage manager Allocate function, or using the Controlled type's Initialize function. The problems with Initialize are: it takes the new object as an "in out" parameter so getting the objects OID requires taking 'Access of the parameter, which is not guaranteed to be its original address; the other problem is that Initialize cannot tell whether the object was allocated by a persistent storage manager or not. The problem with the storage manager Allocate function is that it is used by many different types, so it is more difficult to use it to maintain type extents. It also does not take arguments, so it cannot be used for initialization.

"Access" vs. "in out"

All of the functions in the generated code declare the "self" parameter (the controlling parameter) as an access parameter. The alternative would be to declare the controlling parameter as "in out". There are two reasons why we did not use "in out". The first is that the function will often need to get at the run-time OID of the object, in order to add it into some relationship. With non-tagged types, it is not guaranteed that you can get at the original address of a parameter that is passed in as "in out". The other reason is that persistent objects will usually be held in access variables, and it would be inconvenient to have to use ".all" for all the object parameters on every call. The most common case for a collection to not be held in an access variable is if it is an embedded collection in another object. In that case, it must be declared to be aliased, and then it can be passed using 'Access.

Relationships

Relationships are represented by attributes of the component objects. For a one-to-many relationships, one type has a collection attribute to represent the to-many direction, and the other type has a persistent access attribute to represent to to-one direction. A relationships may be represented only in one direction, or it may be represented in both. When the relationship is represented in both directions, the integrity of relationship is maintained automatically.

There are a number of ways to handle maintaining the integrity of the relationships. It may be possible for the language binding not to address the issue. Any violations of a relationships integrity can be detected and corrected by the ODB at commit time. The problem with this approach is that it means the relationship will be invalid for part of the transaction, since it is only fixed at commit time. Other operations in the transaction may need to be able to depend on the integrity of relationships throughout transactions.

In order to constantly maintain relationship integrity, the language binding needs to be involved. For the Ada binding, we make use of the fact that there is a generator for the Ada code corresponding to the schema. The body of the code for modifying relationships can be generated so that it maintains both directions. For example, from the example at the end of this document, a relationships for marriage is modify by a call to the Marriage procedure. This procedure is generated to look like:

```
procedure Marriage(Self: access Person; S: Person_Ptr) is
begin
   Self.Spouse := S;
   S.Spouse := Self;
end Marriage;
```

With this code for updating a relationship, relationship integrity is maintained.

The problem with this code is that it only works for the to-one direction. The to-many direction of a relationship is represented by one of the collection types, so no new code is generated for modifying the to-many direction of relationships. However, it is possible to create versions of the generic collections that are exclusively for use in relationships. These collections take additional generic parameters that allow the other direction of the relationship to be updated. Below is a generic One_To_Many package that is based on the Set package. Sets are the best basis for One_To_Many relationships, since such relationships only make sense with collections that allow an element to be in the collection at most once.

```
generic
   with package Collection_Pkg is
      new Collection_Package(<>);
   use Collection_Pkg;
   type Owner_Type is private;
   procedure set_backptr(member: access T; new_owner: Owner_Type);
```

```
package One_To_Many is

   -------- One_To_Many ---------

   type One_To_Many(Owner: Owner_Type) is new Set with null record;

   -- functions that need to be overridden for One_To_Many
   -- in order to update the back_ptr

   procedure Insert_Element(Self: in out One_To_Many; E: T);
   procedure Remove_Element(Self: in out One_To_Many; E: T);

end Set_Package;
```

Example

In the following sections we will use the Ada binding to implement the small example that is used in the ODMG specification. Since the Ada binding uses the language neutral ODL to define the schema, we first present the example schema in ODL. After that we present the Ada package specification that corresponds to that schema, followed by the schema package body. Finally, we show a small application that uses this schema. In evaluating the ease of use of this binding, we recommend that you give most of your consideration to what this application code looks like, since it is not generated and so is the kind of code that most application programmer time would be spent with.

We will not give a textual commentary on each of the following files, but will just present them in their entirety. There are some comments in the code explaining some of the more important constructs.

The schema definition

```
// schema.odl: Object Defintion Language schema for People/Cities example

typedef struct {
    integer number;
    string street;
    City city;
} Address_Type;

interface Person {
public:
    extent people;

    attribute string name;
    attribute Address_Type address;
    relationship Person spouse          inverse Person::spouse;
    relationship List<Person> chilren   inverse Person::parents
    relationship Person[2] parents      inverse Person::chilren;
    void birth(in Person new_child);
    void marriage(in Person new_spouse);
    void ancestors(out Set<Person> all_ancestors);
    void move(in Address new_address);
}

typedef Integer City_Code_Type;

interface City {
    extent cities;
    key city_code;

    attribute City_Code_Type city_code;
    attribute string name;
    attribute Set<Person> population;
};
```

The schema package specification

```
with ODB;

package Schema is

   Persistent_Storage : ODB.Persistent_Storage_Manager;
   -- This is the object that will be the storage manager for ALL
   -- persistent types in any database used in this execution.

   -------- Object Types

   type City is tagged limited private;
   type City_Ptr is access all City'Class;
   for City_Ptr'Storage_Manager use Persistent_Storage;

   type Person is tagged limited private;
   type Person_Ptr is access all Person'Class;
   for Person_Ptr'Storage_Manager use Persistent_Storage;

   ------- Types needed for attributes & relationships

   subtype Person_Name_Type is String(1..80);

   subtype Street_Type is String(1..80);
   type Address_Type is record
      Number: Integer range 0..99_999;
      Street: Street_Type;
      City: City_Ptr;
   end record;

   package Person_Collection is new Odb.Collection_Package(Person_Ptr);

   package Person_List is new ODB.List_Package(Person_Collection);
   type Person_List_Ptr is access Person_List.List;
   for Person_List.List_Ptr'Storage_Manager use Persistent_Storage;

   type Person_List_Any_Ptr is access all Person_List.List;
      -- This type used to return embedded lists, like Population(City)

   type Parents_Array is array(1..2) of Person_Ptr;

   package Person_Set is new ODB.Set_Package(Person_Collection);
   type Person_Set_Ptr is access Person_Set.Set;
   for Person_Set_Ptr'Storage_Manager use Persistent_Storage;

   type Person_Set_Any_Ptr is access all Person_Set.Set;
   -- This type used to return embedded sets, like Children(Person)


   ---------------- PERSON ------------------

   -- Extent retrieval function

   function People return Person_Set_Ptr;

   -- Extent name

   Person_Extent_Name : String := "People";
```

```
    -- Create routine

    function Create(Name: String) return Person_Ptr;

    -- Attributes (all public in this example)

    function      Name(Self: access Person) return Person_Name_Type;
    procedure Set_Name(Self: access Person; V: String);

    function Address(Self: access Person) return Address_Type;

    -- Relationships

    function Spouse(Self: access Person) return Person_Ptr;

    function Children(Self: access Person) return Person_List_Any_Ptr;
    -- This returns a pointer to the Children attribute of Person.
    --    It returns the pointer so that changes to it affect
    --    the Person's children list (rather than copy semantics)

    function Parents(Self: access Person) return Parents_Array;
    procedure Set_Parents(Self: access Person; V: Parents_Array);

    -- Operations

    procedure Birth(Self: access Person; Child: Person_Ptr);
    -- a new child is born

    procedure Marriage(Self: access Person; S: Person_Ptr);
    -- a spouse for this Person

    function Ancestors(Self: access Person) return Person_Set.Set;
    -- returns ancestors of this Person

    procedure Move(Self: access Person; New_Address: Address_Type);
    -- moves this person to a new Address

------- CITY ---------

    package City_Collection is new Odb.Collection_Package(City_Ptr);

    package City_Set is new ODB.Set_Package(City_Collection);
    type City_Set_Ptr is access City_Set.Set;
    for City_Set_Ptr'Storage_Manager use Persistent_Storage;

    type City_Code_Type is new Integer;
    subtype City_Name_Type is String(1..80);

    function Cities return City_Set_Ptr;

    function Create(Code: City_Code_Type;
                    Name: String)
                    return City_Ptr;

    function      Name(Self: access City) return City_Name_Type;
    procedure Set_Name(Self: access City; V: String);

    function      City_Code(Self: access City) return City_Code_Type;
    procedure Set_City_Code(Self: access City; V: City_Code_Type);
```

```
   function Population(Self: access City) return Person_Set_Any_Ptr;

   -- Oql instantiation

   function Oql(Query: String) return Person_Ptr;

   -- Extent name

   City_Extent_Name : String := "Cities";

   private
      type Person is tagged limited record
         -- Attributes (all public for this example)
         Name: Person_Name_Type;
         Address: Address_Type;

         -- Relationships
         Spouse: Person_Ptr;

         Children: aliased Person_List.List;
         Parents: aliased Parents_Array;
      end record;

      type City is tagged limited record
         -- Attributes
         City_Code: City_Code_Type;
         Name: City_Name_Type;

         Population: aliased Person_Set.Set;
      end record;

end Schema;
```

The schema package body

```
package body Schema is

   -------- PERSON ----------

   -- Extent retrieval function

   People_Cache: Person_Set_Ptr;

   function People return Person_Set_Ptr is
      function Lookup_P_Set is new ODB.Lookup_Object(Person_Set.Set,
                                                      Person_Set_Ptr);
   begin
      if People_Cache = null then
         People_Cache := Lookup_P_Set(ODB.Default_DB(Persistent_Storage),
                                      "people");
      end if;
      return People_Cache;
   end People;


   -- Creation routine

   function Create(Name: String) return Person_Ptr is
      P: Person_Ptr;
   begin
      P := new Person;
      P.Name := Name;

      Person_Set.Insert_Element(People.all, P);
         -- Can't be done in an Initialize routine, since
         -- Initialize doesn't have an access value for the new object
         -- So: ONLY USE CREATE to make new Person objects
      return P;
   end Create;

   -- Attributes

   function Name(Self: access Person) return Person_Name_Type is
   begin
      return Self.Name;
   end Name;

   procedure Set_Name(Self: access Person; V: String) is
   begin
      Self.Name := V;
   end Set_Name;

   function Address(Self: access Person) return Address_Type is
   begin
      return Self.Address;
   end Address;

   -- Relationships

   function Spouse(Self: access Person) return Person_Ptr is
   S: Person_Ptr;
begin
   return Self.Spouse;

end Spouse;

function Children(Self: access Person) return Person_List_Any_Ptr is
   -- Return pointer to the list of children thats embedded in Person
begin
   return Self.Children'access;
end Children;

-- Operations

function Parents(Self: access Person) return Parents_Array is
   -- Convert array of refs to array of ptrs and return
begin
   return Self.Parents;
end Parents;

procedure Set_Parents(Self: access Person; V: Parents_Array) is
   -- Set new parents (could have been done one parent at a time)
begin
   Self.Parents := V;
end Set_Parents;

procedure Birth(Self: access Person; Child: Person_Ptr) is
begin
   Person_List.Insert_Element_Last(Children(Self).all, Child);
   if Self.Spouse /= null then
      Person_List.Insert_Element_Last(Children(Spouse(Self)).all,
                                      Child);
   end if;
   Child.parents(1) := Person_Ptr(Self);
   Child.parents(2) := Spouse(Self);
end Birth;

procedure Marriage(Self: access Person; S: Person_Ptr) is
begin
   Self.Spouse := S;
   S.Spouse := Self;
end Marriage;

function Ancestors(Self: access Person) return Person_Set.Set is
   -- Returns set of ancestors of Self
   -- Note that the set is not allocated on the heap as it was in C++
   The_Ancestors : Person_Set.Set;
begin
   for I in Self.Parents'range loop
      if Self.Parents(I) /= null then
         Person_Set.Insert_Element(The_Ancestors,Self.Parents(I));
         Person_Set.Union_With(The_Ancestors,
                               Ancestors(Self.Parents(I)));
      end if;
   end loop;
   return The_Ancestors;
end Ancestors;

procedure Move(Self: access Person; New_Address: Address_Type) is
   -- Updates the address attribute of the Person
begin
```

```
   if Self.Address.City /= null then
      Person_Set.Remove_Element(Population(Self.Address.City).all,
                                Self);
   end if;
   Person_Set.Insert_Element(Population(New_Address.City).all, Self);
   Self.Address := New_Address;
end Move;

function Person_Oql is new ODB.Oql(Person,Person_Ptr);
function Oql(Query: String) return Person_Ptr renames Person_Oql;

----- City ------


-- Extent retrieval function

Cities_Cache: City_Set_Ptr;

function Cities return City_Set_Ptr is
   function Lookup_City_Set is new ODB.Lookup_Object(City_Set.Set,
                                                      City_Set_Ptr);
begin
   if Cities_Cache = null then
      Cities_Cache :=Lookup_City_Set(ODB.Default_DB(Persistent_Storage),
                                      "cities");
   end if;
   return Cities_Cache;
end Cities;

-- Creation

function Create(Code: City_Code_Type;
                Name: String)
                return City_Ptr
is
   C: City_Ptr;
begin
   C := new City;
   C.City_Code := Code;
   C.Name := Name;
   City_Set.Insert_Element(Cities.all,C);
   return C;
end Create;

-- Attributes

function Name(Self: access City) return City_Name_Type is
begin
   return Self.Name;
end Name;

procedure Set_Name(Self: access City; V: String) is
begin
   Self.Name := V;
end Set_Name;

function City_Code(Self: access City) return City_Code_Type is
begin
   return Self.City_Code;
end City_Code;
```

```
   procedure Set_City_Code(Self: access City; V: City_Code_Type) is
   begin
      Self.City_Code := V;
   end Set_City_Code;

   -- Relationships

   function Population(Self: access City) return Person_Set_Any_Ptr is
   begin
      return Self.Population'access;
   end Population;

   -- No operations

end Schema;
```

A small application

```
-- Application that loads then consults Person/City database

with Schema;
with ODB;   -- for Database, Transaction, Persistent_Storage...
with Text_IO;

procedure App
  -- The main procedure for this example application
is

   procedure Load (Database : in out ODB.Database) is

      --Transaction which populations the database

      Load_Trans: ODB.Transaction;
      God, Adam, Eve: Schema.Person_Ptr;
      Paradise: Schema.Address_Type;
      People: Schema.Person_Set_Ptr;
      Cities: Schema.City_Set_Ptr;
      procedure Set_Object_Name is new
        ODB.Set_Object_Name(Schema.Person_Set.Set,Schema.Person_Set_Ptr);
      procedure Set_Object_Name is new
        ODB.Set_Object_Name(Schema.City_Set.Set,  Schema.City_Set_Ptr);
   begin
      ODB.Begin_Transaction(Load_Trans);

      -- create both Persons and Cities extension, and name them

      Set_Object_Name(Database, People, Schema.Person_Extent_Name);
      Set_Object_Name(Database, Cities, Schema.City_Extent_Name);

      -- Construct 3 persistent objects from class Person.

      God := Schema.Create("God");
      Adam := Schema.Create("Adam");
      Eve := Schema.Create("Eve");

      -- Construct Address structure, Paradise, as (7 Apple street, Garden)
      -- and set the address attributes of Adam and Eve.

      --* Create overloaded for returning a City
      Paradise := (7, "Apple", Schema.Create(0,"Garden"));

      Schema.Move(Adam,Paradise);
      Schema.Move(Eve,Paradise);

      -- Define the family relationships

      Schema.Birth(God,Adam);
      Schema.Marriage(Adam,Eve);
      Schema.Birth(Adam, Schema.Create("Cain"));
      Schema.Birth(Adam, Schema.Create("Abel"));

      -- Commit transaction, thus putting these objects into the database
      ODB.Commit_Transaction(Load_Trans);
   end Load;
```

```
procedure Print_Persons(S: Schema.Person_Set.Set)
  -- A service function to print a set of Persons
is
   P : Schema.Person_Ptr;
   I : Schema.Person_Collection.Iterator'Class
     := Schema.Person_Set.Create_Iterator(S, True);
begin
   while Schema.Person_Collection.More(I) loop
      Schema.Person_Collection.Next(I, P);

      Text_IO.Put("--- " & Schema.Name(p) & "lives in ");
        if Schema.Address(P).City /= null then
        Text_IO.Put_Line(Schema.Name(Schema.Address(P).City));
        else
           Text_IO.Put_Line("unknown");
        end if;
   end loop;
   Schema.Person_Collection.Delete(I);
end Print_Persons;

procedure Consult (Database : in ODB.Database) is

   -- Transaction which cosults and updates the database

   Consult: ODB.Transaction;
   Adam, Abel: Schema.Person_Ptr;
   Earth: Schema.Address_Type;
   S: Schema.Person_Set.Set;
begin
   ODB.Begin_Transaction(Consult);
   -- Begin the transaction

   Text_IO.Put_Line("All the people...:");
   Print_Persons(Schema.People.all);

   Text_IO.Put_Line("All the people sorted by name...:");
   Schema.Person_Set.Oql(S, "sort p in people by p.name");
   Print_Persons(S);

   Text_IO.Put_Line(
           "People having 2 children and living in Paradise...:");
   Schema.Person_Set.Oql(S,
                        "select p from p in people " &
                        "where p.address.city != nil " &
                        "and p.address.city.name = ""Garden"" " &
                        "and count(p.children) = 2");
   Print_Persons(S);

   -- Adam and Eve are moving ...
   Earth := (13, "Macadam", Schema.Create(1, "St-Croix"));
   Schema.Oql(Adam, "element(select p from p in people " &
                   "where p.name = ""Adam"" ");
   Schema.Move(Adam,Earth);
   Schema.Move(Schema.Spouse(Adam),Earth);

   Text_IO.Put_Line("Abel's ancestors...:");
   Abel := Schema.Person_List.
             Retrieve_Element_At(Schéma.Children(Adam).all,0);
   Print_Persons(Schema.Ancestors(Abel));
   ODB.Commit_Transaction(Consult);
   end Consult;

begin -- App Main program
   declare
      Database: ODB.Database;
   begin
      ODB.Open(Database,"family");
      ODB.Set_Default_DE(Schema.Persistent_Storage, Database);
      Load(Database);
      Consult(Database);
      ODB.Close(Database);
   end;
end App;
```

Implementing the ODMG binding using Texas

We implemented this ODMG binding using Texas as the backend storage manager. In some respects Texas is not truly an object-oriented database manager, since it does not support multiple users with concurrency control, nor does it have any declarative query capability. However, it does have almost all of the important features necessary to test this binding. It uses pointer swizzling, like ObjectStore, for persistence and it supports databases, transactions, logging and recovery.

One advantage of using Texas is that, because it uses pointer swizzling for persistence, it was not necessary to modify an Ada compiler to support the 'Storage_Manager attribute. Recall that the primary motivation for the new storage manager attribute is that the storage pool facility is not sufficient to support non-swizzling object-oriented databases, since it does not have a hook into dereference. So, we were able to create a complete implementation of the ODB package based only on the storage pool capabilities already available in Ada.

We used Gnat as our Ada 95 compiler. Gnat worked well for us, since it is a complete implementation of Ada 95, it is fairly stable, and it supports storage pools. We did have a few problems that could be tied to the combination of using Texas without storage managers and using the Gnat compiler. Unfortunately, Gnat represents pointers to arrays as "fat" pointers. Texas was written for use with C++, so it could not swizzle or return a fat pointer. Because of this, we were forced to create alternate versions of the persistent storage pool, depending on whether it was storing objects, fixed length arrays of objects, or unconstrained arrays of objects.

We did encounter some problems based on the fact that we were using storage pools instead of storage managers. Recall that in the specification of storage managers, we did not have them inherit directly from storage pool. This was because we needed the Allocate routine to take an extra parameter: the string representing the type being allocated. Almost all object-oriented database systems, including Texas, need to know what type they are allocating. Unfortunately, the way Allocate is defined for the Storage_Pool type, the only information that is passed about the type is its size.

The way we got around this was to add a new discriminant to the ODB_Persistent_Pool type. Then, for each access type that is to be made persistent, the programmer must both create a new instance of the ODB_Persistent_Pool type, using the designated type name as the discriminant, and add a representation clause so the access type uses that storage pool. This way each access type gets its own storage pool that knows what type it is allocating. Unfortunately, in addition requiring an extra line of code for every access type declaration, it also makes it harder to write some generics, since generics often need to define new access types and give them the same persistence as a generic parameter. This would be possible if it were possible for different access types to share a storage pool. Without that ability, generics that can be used for both transient and persistent instantiations become more difficult. For more information on how we accomplished this, see the later section on transitioning MOO to become persistent.

The following is the modified version of the ODB specification, modified to work with Gnat and Texas using only storage pools:

```
with System.Storage_Pools;
with System.Storage_Elements;
with Unchecked_Deallocation;
with Interfaces.C.Strings;

package ODB is

    package CStr renames Interfaces.C.Strings;
```

```
-- helper types

type Access_Status is (Not_Opèn, Read_Write, Read_Only, Exclusive);
type Void_Ptr is private;

---------------- Database ----------------

type Database is limited private;

-- Most operations dealing with the database assume that
-- they are operating on the Default_Db.
procedure Set_Default_Db(Db: Database);
function Default_Db return Database;

procedure Open(Self   : in out Database;
           Name   : String;
           Status : Access_Status := Read_Write);
procedure Open(Name   : String;
           Status : Access_Status := Read_Write);
procedure Close(Self : in out Database);
procedure Close;

generic
  type Object_Type(<>) is limited private;
  type Object_Ptr is access Object_Type;
procedure Set_Object_Name(Object : Object_Ptr;
                  Name   : String);
generic
  type Object_Type is limited private;
  type Object_Ptr is access Object_Type;
function Get_Object_Name(Self   : Database;
                  Object : Object_Ptr) return String;

generic
  type Object_Type(<>) is limited private;
  type Object_Ptr is access Object_Type;
function Lookup_Object(Name : String) return Object_Ptr;

generic
  type Array_Type(<>) is limited private;
  type Array_Ptr is access Array_Type;
procedure Set_Unconstrained_Array_Name(
  Array_Object : Array_Ptr;
  Name         : String
);
generic
  type Array_Type(<>) is limited private;
  type Array_Ptr is access Array_Type;
function Lookup_Unconstrained_Array(Name : String) return Array_Ptr;

---------------- Transaction ----------------

type Transaction is limited private;

procedure Begin_Transaction     (Self : in out Transaction);
procedure Commit_Transaction    (Self : in out Transaction);
procedure Abort_Transaction     (Self : in out Transaction);
procedure Checkpoint_Transaction(Self : in out Transaction);
```

```
-------------- STORAGE POOL TYPE -------------------
subtype CString is CStr.chars_ptr;

type Type_Name_Ptr is access String;

type ODB_Persistent_Pool(Type_Name: access String) is abstract new
  System.Storage_Pools.Root_Storage_Pool with
record
  C_Type_Name: CString := CStr.New_String(Type_Name.all);
  Texas_Unique_Cached_Type_Id : Integer := -1;
  Texas_Unique_Cached_Hash_Val: Integer := -1;
end record;

-------------- Inheritable Storage Pool Routines ----------------

function Storage_Size (Pool : ODB_Persistent_Pool)
  return System.Storage_Elements.Storage_Count;

procedure Deallocate
  (Pool         : in out ODB_Persistent_Pool;
   Address      : System.Address;
   Storage_Size : System.Storage_Elements.Storage_Count;
   Alignment    : System.Storage_Elements.Storage_Count);

type ODB_Pool_Type(Type_Name: access String) is new
  ODB_Persistent_Pool(Type_Name) with null record;

type ODB_Array_Pool_Type(Type_Name: access String) is new
  ODB_Persistent_Pool(Type_Name) with null record;

type ODB_Fixed_Length_Array_Pool_Type(Type_Name: access String) is new
  ODB_Persistent_Pool(Type_Name) with null record;

type ODB_Abstract_Pool_Type(Type_Name: access String) is new
  ODB_Persistent_Pool(Type_Name) with null record;
```

One large advantage to using Gnat was the way it handles tagged types. Gnat uses the same virtual function table (VFT) mechanism that is used by Gnu C++. The only difference is that Gnat stores the VFT field as the first field of the record, whereas Gnu C++ stores it as the last field of the top-most ancestor of the record.

Texas needs to know about the VFT field in order to correctly handle it at swizzle time. The VFT field points to an array of pointers to functions. Because functions are not actually stored in the database, Texas cannot swizzle the VFT the way it does other pointer fields. Instead it needs to store the fact that there is a VFT field in that record for that type. Then when the data is being restored, Texas notes that there is a VFT field and finds the table based on its knowledge of where all the VFTs are stored.

The only thing we needed to do to get the VFT swizzling to work correctly was to modify the way Texas looks for VFT fields. For both GNU C++ and Gnat, the field is signaled by a naming convention in the debugging symbol output for the record. We just needed to look for the different naming convention. (Having source code to the object-oriented database system was, of course invaluable. If we were trying to implement this with ObjectStore, we probably would have needed a lot of help from the company).

Another advantage to using Gnat with Texas has to do with the way Texas gets type information from an application. Texas contains a modified version of the Gdb debugger. The debugger reads the debugging information about the application and builds up an internal representation of the types in the application. Texas uses that internal representation to construct its own representation of the persistent types, which it then stores to disk in the TDF file (for Type Definition File). Since Gnat writes out debugging information that can be read by Gdb, Texas is able to build an accurate representation of the types. There were only a few problems. We had to make some minor modifications to this schema generation tool so that it recognized Ada's built-in types, since they don't have any information in the debugging output. We also had to fix it to understand Ada arrays, which had the side benefit that it made it possible to understand Ada strings. The last problem was discriminated records. Discriminated records were difficult because, not only didn't Gdb handle them, but Gnat didn't even output any debugging information for them. What we did was create a separate Ada file called discriminated_type_info.adb, which contained alternate versions of those types that were undiscriminated. We then compile that new file and use its definitions of the types.

Another change we had to make to Texas was in how it finds the persistent types in the application. For C++, Texas searches each source file for calls to "new", and then notes the type that is being created (all at compile time). This way the schema may be smaller, since it is not necessary to keep information about every type, just those that might become persistent. We modified this algorithm to search the source files for

Storage_Pool declarations.

Demonstration Application: MOO

In order to determine how useful our binding was for creating large persistent Ada applications, we knew it would be necessary to create a non-trivial example application. We decided to build a MOO (for Mud Object-Oriented) server. We started building the application even before we had finished the implementation of the binding for Texas, so that we could modify the binding according to issues that came up from the development of the MOO.

We wanted to see how difficult it would be to turn a transient application into a persistent application. So, our first implementation of MOO was transient. We did, however, use the ODB collection classes even in our first implementation. In this respect the example could not be used to examine how difficult it would be to transition from using some other collection library (or even just using arrays for all collections) to using ODMG collections. Nonetheless, it is a valid test for determining how easy it is to make an existing transient application persistent, since it is not necessary for an application to make use of the ODMG collection classes in order to become persistent. And, in fact, we did have a number of places in our code which used plain arrays instead of collections.

By using collections, however, we were able to determine how useful they were, and find ways to improve them. It is through this testing that we discovered that multiple "use" clauses for collections caused colliding function definitions, which we were able to fix by introducing the Collection_Ancestor type. We also discovered that Ada arrays were a good way for dealing with collections, so we added routines for getting and inserting arrays from collections.

Making MOO Persistent

Once we had a transient implementation of MOO, we undertook to finish the implementation of Texas, and to make whatever modifications to MOO were necessary to make its data persistent. The following modifications were required:

Adding 'Storage_Pool clauses

The main step in the process was to go to each of the access types that were to be made persistent, and add a representation clause of the form:

```
for MOO_Value_Ptr'Storage_Pool use Persistent_Storage_Pool
```

Convert global variables to root objects

The global variables in the transient program were assumed to be initialized either during elaboration or within an initialization routine, and then accessible for the rest of the system lifetime. The system now had to be able to shutdown and restart and still be able to access those variables. So, the initialization of these variables had to be divided into the first time initialization and restart initialization. When the application was being run with an existing database, the program had to use the Lookup_Object_Name function to get at the global object. If there was no such object, then that could be used as a signal that it was a new database, and the object could be created and stored using Set_Object_Name.

Unconstrained array roots

Some of the global variables were pointers to arrays. Because Gnat uses "fat" pointers to represent pointers to unconstrained arrays, we discovered that we needed to add new routines for setting and getting root objects that were unconstrained arrays. These special routines created a wrapper object that contained one field which was the pointer to the unconstrained array. Lookup_Object_Name would then just get the wrapper object, and then return its one field.

Eliminate non-access globals

The only kind of global variables that can be stored in the database are access types. Any integers, strings or other scalar values that were stored in global variables could only be stored in the database if they were stored on the heap. For one set of these global variables, we collected all of the global variables for that package and put them in a single object that we stored on the heap. For other global variables in other packages, we just changed them from being scalars to pointers to scalars. Recall that with the storage pool mechanism for persistence, it is *not* necessary to put a scalar in an object before storing it in the database.

Separate transient and persistent access types

Some access types in an application could be used to refer to both transient and persistent values. Since the access type determines whether the value is persistent or not, these types have to be divided into persistent versions and transient versions. For our application, there was only one such access type that we had to divide up: String_Ptr. Since the only non-persistent use of it was for error strings, we were able to create a new access type Error_String_Ptr.

Divide some generic packages

For one of the generic packages we created for MOO, the Hash_On_Integer package, we sometimes instantiated it for transient data and sometimes for persistent data. The package needs to define a new access type that is private to the package and which has to be persistent if the contents of the hash table are persistent, and transient if the contents are transient. If it were possible to share the same storage pool object for all of the persistent access types this would be easy. There would be a representation clause in the generic stating that it should use the same storage pool as the access type passed in, e.g.:

```
for Hash_Element_Ptr'Storage_Pool use Element_Ptr'Storage_Pool;
```

We cannot share storage pools, however, because, as explained above, it is necessary to store the name of the designated type as a discriminant of the storage pool. So, we were forced to create two generic packages where there was previously only one: Persistent_Hash_On_Integer and Transient_Hash_On_Integer. The persistent version declared and used persistent storage pools, the transient version did not. We did not, however, just create the two packages using cut-and-paste. Instead we created a third package that does all of the work of the two other packages, but leaves the definition of the access type as yet another generic parameter.

No pointers to functions

Other than the virtual function tables implicitly declared in tagged types, none of the persistent objects may contain a pointer to a function. While it is not theoretically impossible to handle making such pointers persistent, we did not implement that logic. We also didn't have any pressing need for making function pointers persistent.

Define nondiscriminated versions of discriminated types

Since Gnat does not write out debugging information about discriminated types, and Texas depends on the debugging information for building up its schema, it was necessary to define versions of the discriminated types that were not discriminated, and put those definitions in a file called "discriminated_type_info.adb". For MOO we had only one such type that had to be made persistent: the MOO_String_Type. The discriminated version was

```
type MOO_String(Length: MOO_String_Length_Type) is new MOO_Value with
record
    Value: String(1..Length);
end record;
```

The nondiscriminated version was:

```
type MOO_String is tagged
record
  Length: Integer;
  Value: String(1..8);
end record;
```

This nondiscriminated version could then be used for the schema. The fact that its length might be wrong was not important, since the size of the object was determined by the size passed in at Allocation time, rather than based on the schema. This schema would only fail to work if there were fields after a field whose size was determined by the discrminant.

No abstract object creation

Another problem with using storage pools instead of storage managers (and therefore not being able to share pools), was that new persistent objects could not be created and assigned into ancestor access types. Since the access type not only determined whether the object was persistent or transient but also told the database what type was being created, it was imperative that the target type for an object creation be of exactly the right access type. So, for example, MOO_Value is the parent of MOO_Number, but we could not have code of the form:

```
V: MOO_Value_Ptr := new MOO_Number;
```

Instead we had to have an intermediate variable that would get the new object, then later assign to the more general variable.

Performance

We compared the performance of our persistent MOO server with LambdaMOO, which was written in C and stores all of its data in virtual memory. We first did a number of small tests that compared specific aspects of our interpreter with the LambdaMOO interpreter. For example we tested a tight loop of property accesses, a tight loop of verb (i.e. function) calls, a loop of certain built-in function calls, etc. These were useful in finding problems where we were considerably slower, usually due to some bug.

The real performance measure, however, was a more comprehensive script. We loaded a database that contains many complex functions and objects. We then devised a script that would make extensive use of the most computationally intensive features of the objects in the database and create new objects and verbs. The script was about 150 commands, and it took around 52 seconds to run on the LambdaMOO server.

To test the speed of our MOO server, we first compiled the whole server with Ada run-time checks turned off and full-optimization on. We could not turn on function in-lining, because the version of Gnat we are using (v. 3.05) doesn't correctly implement it (although the previous version apparently did, as will the next version).

We then ran the server. When you start the LambdaMOO server it takes several minutes for it to start up because it has to load the entire database into virtual memory before it can start accepting commands. With our server, it started in about a second. However, if we then ran the script as the first commands to the server, it took about 90 seconds. The first commands of the script took a long time, as data was being activated from disk, and as the script continued the speed continued to increase.

We then ran the script a second time. Running a second time guarantees that all of the data is off disk and in-memory, so it is an accurate comparison with the LambdaMOO server which also has all of the available data in memory. It ran in 58 seconds. So, we are about 10% slower than the LambdaMOO server when run on data that is already in memory.

Conclusion

Intermetrics has designed and implemented a convenient and effective mechanism for making large complex Ada95 programs persistent. We defined an Ada95 binding to the ODMG object-oriented database standard, and then implemented a version of it using the Texas object-oriented database as the backend.

We then created a large persistent application using our implementation of the ODMG binding. The application implemented a complete MOO server. We found that using the binding made it easy to create a large new application, and we demonstrated that it is easy to transition an application that uses transient data into one that is persistent.

DISTRIBUTION LIST

AUL/LSE Bldg 1405 - 600 Chennault Circle Maxwell AFB, AL 36112-6424	1 cy
DTIC/OCP 8725 John J. Kingman Rd, Suite 0944 Ft Belvoir, VA 22060-6218	2 cys
AFSAA/SAI 1580 Air Force Pentagon Washington, DC 20330-1580	1 cy
PL/SUL Kirtland AFB, NM 87117-5776	2 cys
PL/HO Kirtland AFB, NM 87117-5776	1 cy
Official Record Copy PL/VTS/Capt Lindsay Kirtland AFB, NM 87117-5776	2 cys
PL/VT Dr Hogge Kirtland AFB, NM 87117-5776	1 cy